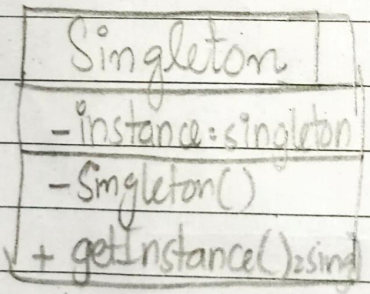
Design Pattern # 01
Singleton Pattern

- Singleton is a creational design pattern that lets you ensure that a class has only **one instance**, while providing a global access point to this instance.
  - → ensures that a class has just a single instance.
  - → provide a global access point to that instance.

* Default constructor → private ...
* static creation methods that acts as a constructor. this will calls the private constructor to create an obj & saves it in a static feild. (returns catched obj)




```
if (instance == null) {
    instance = new Sing()
}
return instance
```

Lazy initialization: make the instance only when needed

Eages initialization: create the instance at the start of program (use ho ya na ho)

Date: Day:

```
class Singleton {
  private:
  static Singleton * instance;
  int data;
//made it private
  Singleton () {
        data = 0;
        }
// A public function that calls constructor -- if no obj made
public:
   static Singleton * getInstance()
        if (instance == NULL) {
            instance = new Singleton();
        }
        return instance;

    }

// Implementation
   Singleton * Singleton :: instance = NULL;
int main () {
     Singleton * s = Singleton :: getInstance();
        ----
        }
```

Date: Day:

child class creates diff product.

Design Pattern # 02
Factory Method Pattern

Provides an interface for creating obj in a superclass but allows subclasses to alter the type of objects that will be created.

→ Virtual constructor.

→ tight coupling violation of OCP.

```
class Pizza {
public:
    void prepare() const {
        cout << "Making Pizza";
    }
};
class Pasta {
public:
    void prepare() const {
        cout << "Making Pasta";
    }
};
class Waiter {
+ void takeOrder(const string& type) {
    if (type == "Pizza") {
        Pizza* pizza;
        pizza->prepare();
    }
    if (type == "Pasta") {
        Pasta pasta;
        pasta.prepare; }
```

else ...

→ (After Applying Factory Method)

Date: Day:

```
class Dish {
    + pure virtual void prepare() const = 0;
};

class Pizza : public Dish {
    void prepare() const override {
        cout << "Preparing Pizza";
    }
};

class Pasta : public Dish {
    + void prepare() const override {
        cout << "Preparing Pasta";
    }
};

class chef {
    virtual Dish * createDish() const = 0;
};

class PizzaChef : public chef {
    + Dish * createDish() const override {
        return new Pizza();
    }
};

class PastaChef : public chef {
    Dish * createDish() const ov {
        return new Pasta();
    }
};
```

high

spring

Lecture #21

Date: 10 Nov 2025 Day Monday

Design Pattern : State Pattern

```
class Book {
    int state;
    bool issue (Member * mem) {
        switch(state) {
        case AVA:
            ------
            return true;
```

A

```
case ISS:
case RES:
    return false;
```

B

```
bool reserve (Member * mem) {
    switch (state) {
    case ISS:
        ------
        return true;
```

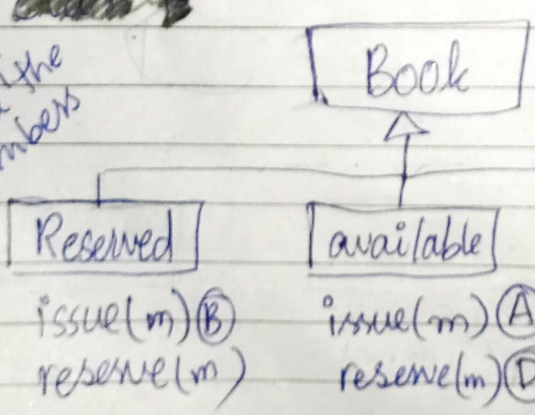
C

```
    case AVA:
    case REF:
        return false;
```

D

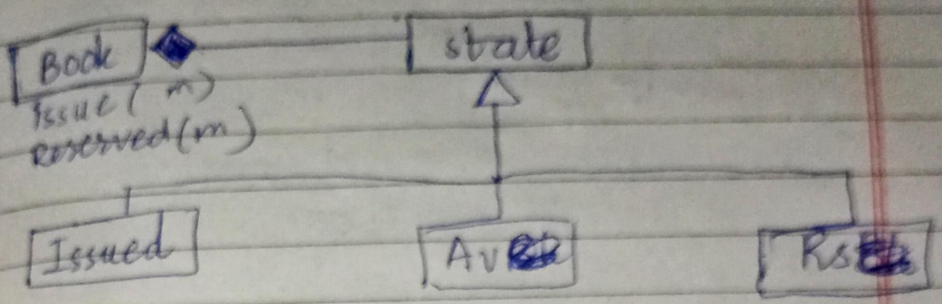
```
        ------
    }
```

* has an issue of copying all the data again & again



issue(m) = 0
reserve(m) = 0

switch case is replaced by polymorphism




```
class Book{
    state * st;
bool issue(member *mem){
    return st->issue(mem);
    }
bool reserve(member * mem){
    return st->reserve(mem);
    }
```

```
class Book{
    state * st;
bool issue(member *mem){
    bool f = st->issue(mem);
    if(f){
        setState(new Is());
    return f;
    }
```

Date: Day:

```
bool reserve (Member * mem){
        bool f = st -> reserve (mem);
    if (f)
        setState (new Rs());
    return f;

    }

void setState (state * s) {

        delete st;
        st = s;

    }
```

Lecture #22

Date: Day:

Design Pattern #04
Observer Design Pattern

Lecture #23

Date: Nov 17, 25 Day: Monday

Composite (obj oriented data structure)

```
struct File {
        char * name;
        int size;
    }

int callSize (file *Fl) {
        return fl -> size;
        }

int calSize ( Folder * fd) {
        int sum = 0.
    for (int i = 0 ; i < M; ++i)
        sum += calSize ( fd -> a[i] )

    for (int j = 0 ; j < N ; ++j)
        if ( fd -> b[j] != NULL)
            sum + = calSize (fd -> b[

        return sum ;

    }

struct Folder {
        char * name ;
        File * a [M] ;
        Folder * b[N] ;

    }
```

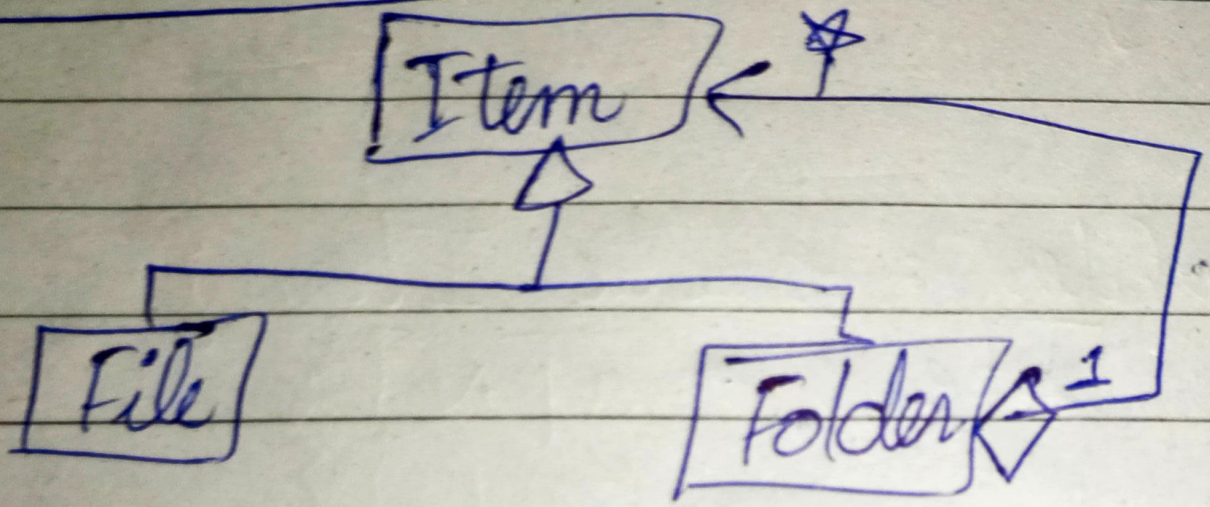
Day:

Date:

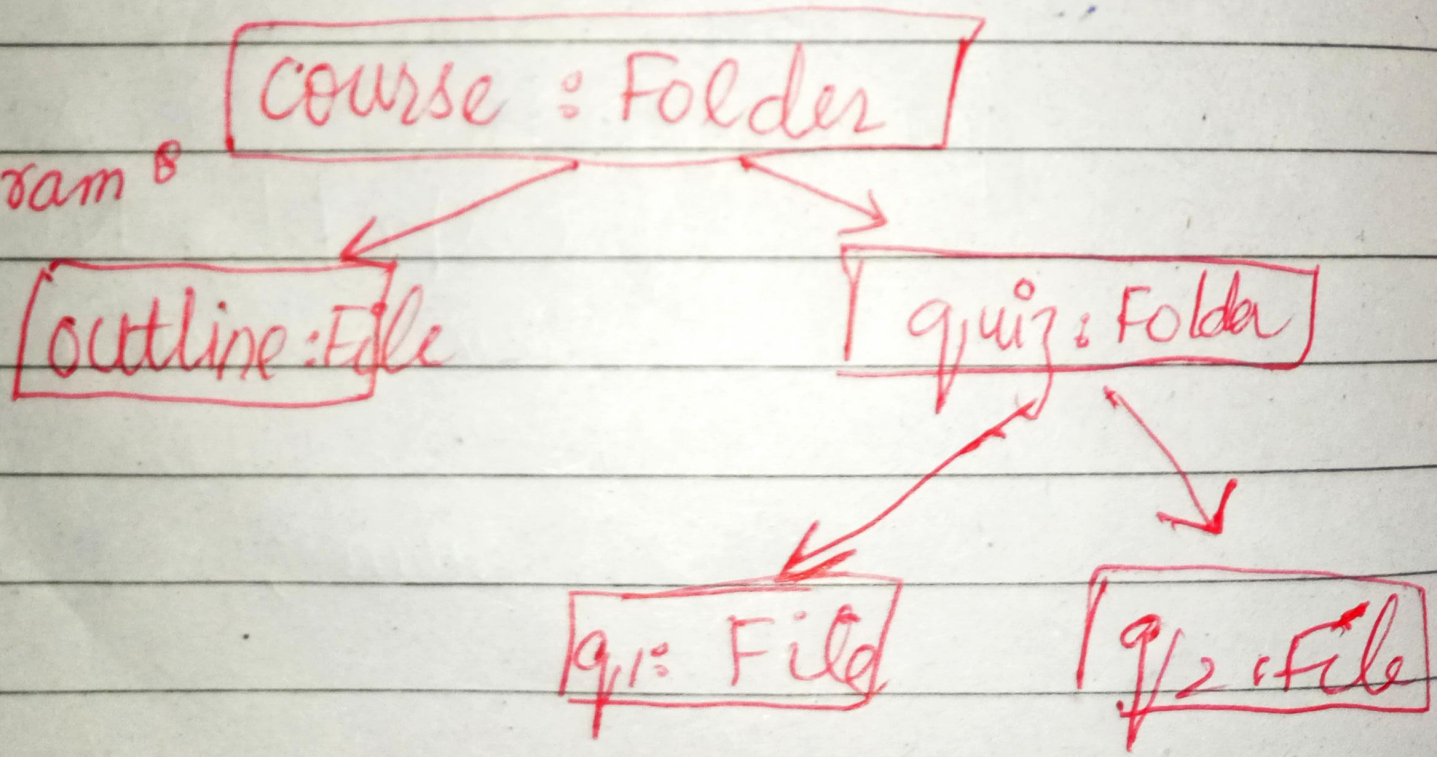
Improved Design:

→ make classes instead of structs
→ Generalized the File & Folder classes.

```
class Item{
    char * name;
    virtual int calSize() = 0
    };

class File : public Item{
    int size;

    int calSize(){
        return size;
    }
};

class Folder : public Item{
    Item * a[N];
    int calSize(){
        int sum = 0;
    for (int i=0; i<N; ++i){
        if (a[i] != NULL)
            sum += (a[i]->calsize());

    return sum;

    }
```



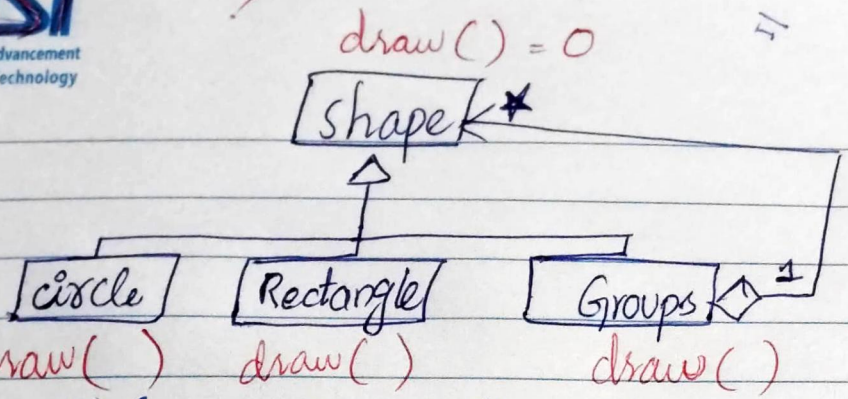
Object Diagram



Foundation for Advancement of Science & Technology

17/11/25

wow!

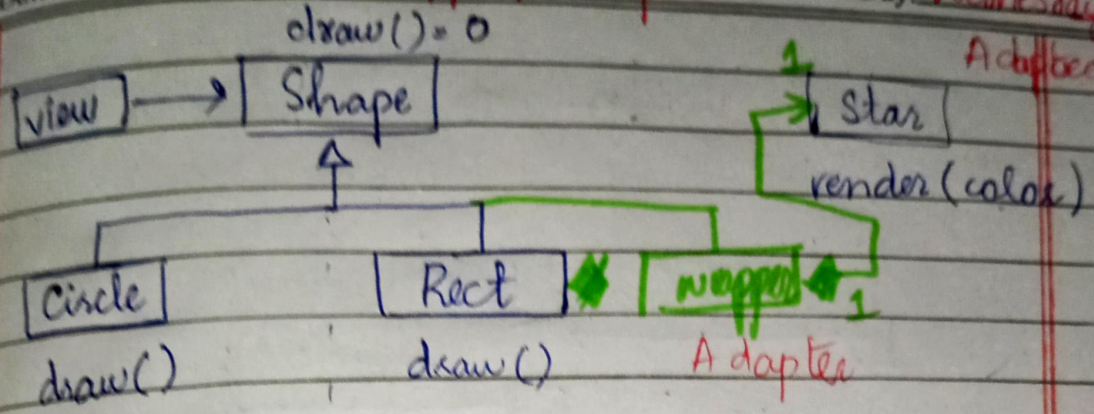


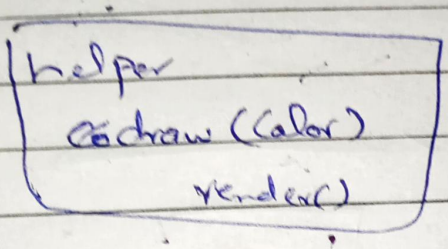

```
Group(){ : public Shape {
    shape * arr[];
draw() {                    'sh'
    for each shape in group
        sh->draw();
}
```

```
class Group : public Shape {
    shape * a[n]

    void draw(){
        for (int i = 0; i < N; i++)
            if(a[i] != null) a[i]->draw();
    }
```

Lecture #24

Date: 19 Nov 25 — Wrapper / Adapter — Day: Wednesday





class ~~Wrapper~~ : public shape {

~~star*~~ star obj

~~color~~

void draw() {

obj.render(BLACK);

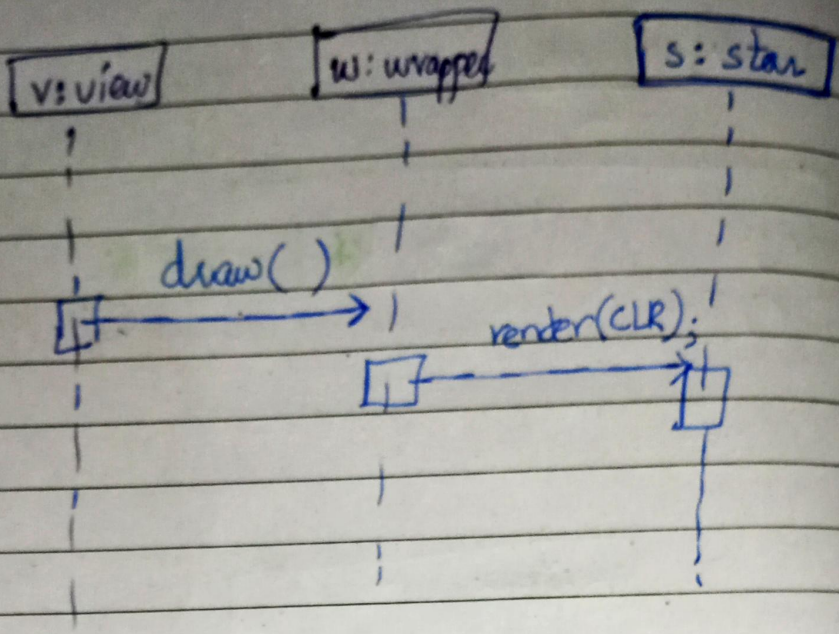
~~obj.render(color)~~

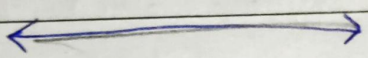
- - - -

}

}

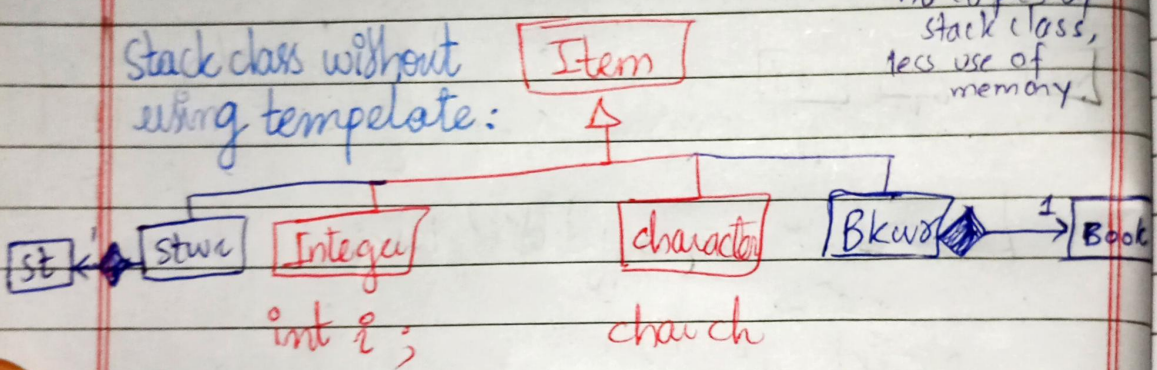
Date:





no copies of stack class, less use of memory.

Stack class without using tempelate:

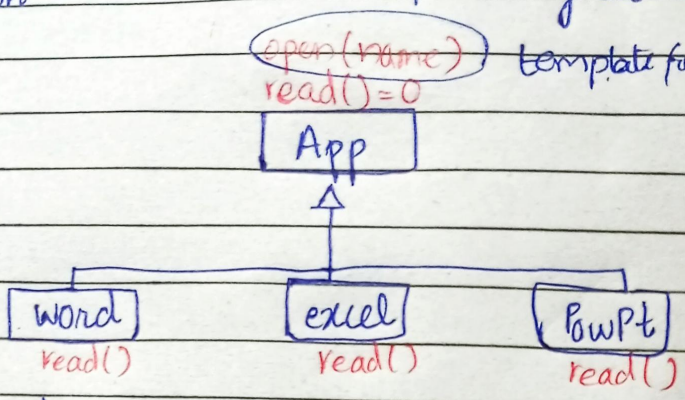



```
class stack {
        Item * a[N];
        int top = 0;

void push (Item *m) {
        a[top] = m
        ++top;
        }
Item* pop ( ) {
        return a[--top];
        }
```

Date: Nov 24, 25

Template method

"This pattern allows you to write parent class code even when you don't known each & every detail."

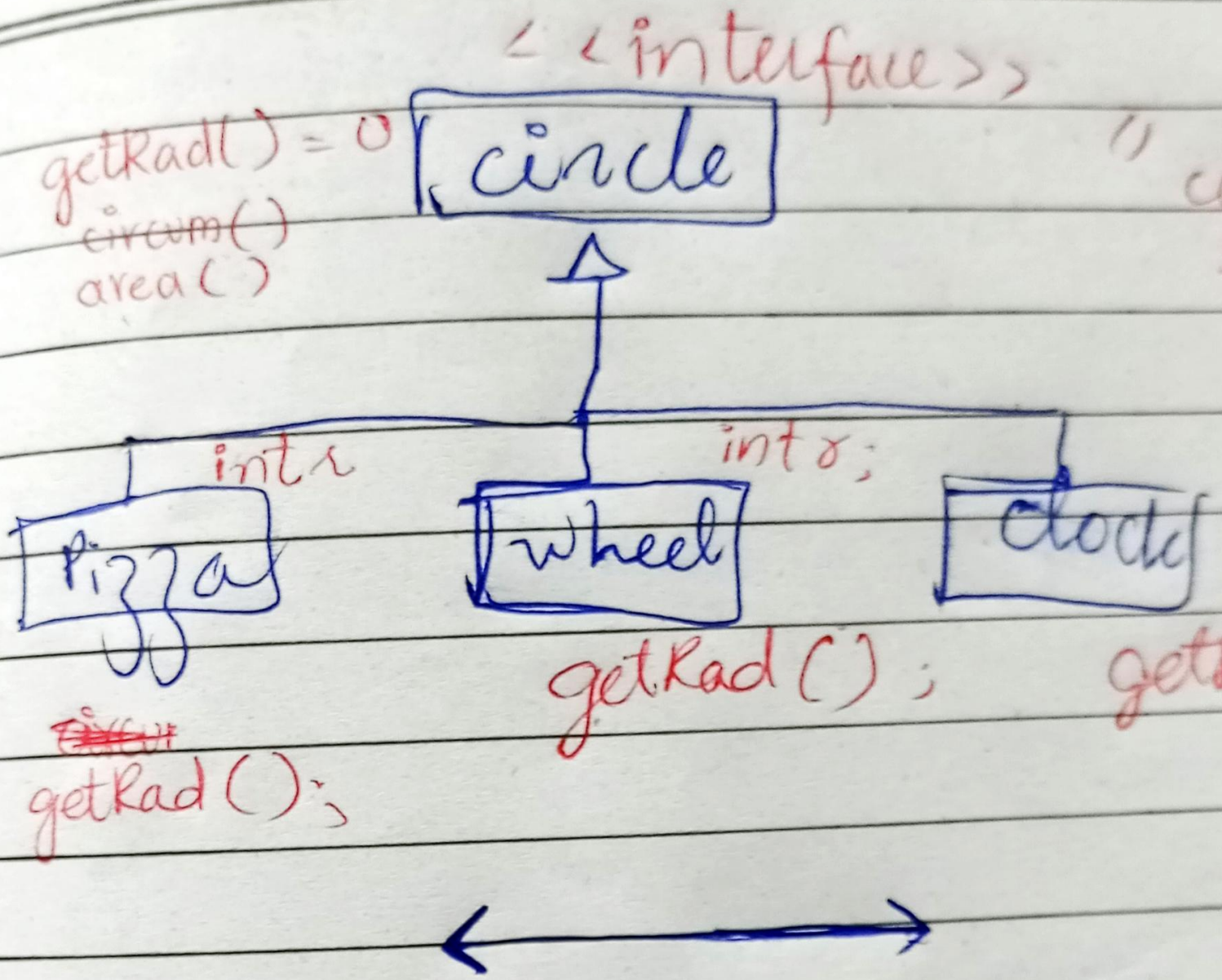



```
class App {
    bool open(char * name){
1. Check file exists
2. Check privilages & rights
3. Ask password if any
4. Show status / progress
5. Call read()
6. Display data.
```

```
interface Circle {
 int circum(------){ return 2π*getRad();
     area(------)
      int getRad() = 0;
 }
```

make a getter function to get the radius of the child classes.

```
int area(){
    int rad = getRad();
    return PI * rad * rad;
}
```


<<interface>>
circle
getRad() = 0
circum()
area()
can be seen as
int r
int r;
int r;
Pizza
wheel
clock
getRad();
getRad()
getRad();

* getGPA() {

**